**Indonesian Journal of Computer Science**
ISSN 2549-7286 (*online*)
Jln. Khatib Sulaiman Dalam No. 1, Padang, Indonesia
Website: ijcs.stmikindonesia.ac.id | E-mail: ijcs@stmikindonesia.ac.id

# AUTOMATIC GENERATION OF UNIT TEST DATA FOR DYNAMICALLY TYPED LANGUAGES

**Hayatou Oumarou[1,2], Faouzi El Mansour[2]**
Oumarou.hayatou@yahoo.fr, elmansourfaaouzi@gmail.com
[1] Department of Computer Science, The Higher Teachers' Training College, University of Maroua, Cameroon
[2] LaRI Lab, University of Maroua, Cameroon

**Article Information**

Submitted : 16 Sep 2023
Reviewed: 29 Sep 2023
Accepted : 30 Sep 2023

**Keywords**

Testing, Unit testing, Automation, Data generation

**Abstract**

Testing is the major means of verifying and validating software. It is a repetitive and time-consuming activity. Testing is neglected because of its high cost and the fact that it does not add functionality to the system. As a result, many programmers don't write tests. To remedy this, some researcher proposed automatic test generation. Test generation is a solution that reduces workload and increases productivity. In this paper, we propose a test data generation approach for unit tests in dynamically typed languages. Our approach is based on the analysis and decomposition of the AST (Abstract Syntax Tree) obtained when compiling the source code of the method under test. We validate this approach in Pharo a real system. The results on three systems show the effectiveness of the approach.

Attribution-ShareAlike 4.0 International License

## A. Introduction

Testing enables developers to be sure that software functionalities comply with customer expectations and are bug-free. It consists of analyzing a program with the intention of detecting anomalies [1]. According to MEYER Bertrand in [2], a software test is carried out to show the presence of one or more bugs, and never their absence. This stage takes up around 40% to 50% of software development effort, and a third of total project time [3]. In addition, this activity is sometimes responsible for introducing errors into the software source code [1]. To address these problems, some authors have proposed test generation. A test generator is a tool that helps developers produce software test data. It must be general and generate data that corresponds to the chosen criteria (coverage, branching...). The language used to write the software strongly influences these solutions. Today, dynamic typing languages are gaining ground in the software industry [4]. Generating reliable test data in these languages is a real challenge, due to the lack of information on variable types. Strategies implemented to automate the unit test data generation have their limitations.

In this paper, we propose a method for semi-automatic generation of unit test data for dynamically typed languages. It is based on analysis and decomposition of the AST to generate test values.

The rest of the article is organized as follows: 2: unit test generation. 3: Proposed approach. 4: Results and discussions. 5: Threats to validity. 6: Conclusion.

## B. Unit test generation

Software testing is the process of analyzing a program with the intention of detecting anomalies. It ensures that the software meets the specifications. [5]. Testing takes place at different levels of granularity. We will focus on unit testing. In the literature, we identify two approaches to unit testing: static and dynamic.

In the static approach, the program is not executed. Static analysis of the code, design documents and algorithms are used to check for errors and ensure that the program is running correctly. [1]. With the dynamic approach, the program is executed. In this case, input data is sent and the results are compared with what is expected to detect errors. In this article, we focus on the dynamic testing approach.

Test data generation involves identifying and selecting input data that meet certain criteria [6]. Many techniques have been developed in the literature to automatically generate this data for unit tests [7]. We identify three ways of generating data for unit testing. We have random generation, static generation and dynamic generation.

- Random test data generation : is a technique that generates data according to requirements, specifications or other conditions defined by the tester. It selects test data at random from the set of all possible inputs [8]. Random generation methods include Adaptive Random (AR) [9]Feedback Directed Random(FDR) [10]. One of the limitations of these techniques is the invalid domain: there is no guarantee that the values generated as input to a program under test will be accurate. The data generated may be too close or too far from the input.

- Static test data generation technique: this is a form of test data generation without actually running the software. It relies on requirements documents, software source code and design schemas to generate data either manually or automatically. [11]. The static approach to test data generation begins by generating a Control Flow Graph (CFG). These include symbolic execution and domain reduction. The main limitation is that, in the case of a loop with a variable number of iterations, we end up with an infeasible path. It is therefore difficult to solve a problem described in the form of constraint system, where the conditions required to traverse a path do not exist.
- Dynamic test data generation technique: Dynamic test data generation is based on the analysis of actual SUT execution. In contrast to the static generation approach, this approach consists of executing the system under test on the basis of concrete data supplied by the user. Instead of using variable substitution, it executes the system under test with certain inputs, possibly chosen at random. It is not bounded like symbolic execution. The behavior of the SUT during its analysis can reveal whether the path taken is the right one or not. If the path taken is not the right one, backtracking is used to find the node where the flow took the wrong path. These techniques include intelligent approach [12] search-based approach [13].

In practice, static generation may be combined with symbolic execution, dynamic generation and/or concrete execution to generate test data. This combination is called concolic (conc(rete+symb)olic) [14].

Automatic generation of unit test data for dynamically typed languages presents certain challenges. The lack of type information is a major obstacle to automation. To circumvent this problem, approaches have been proposed by some researchers to obtain the types of variables in a program written in these languages. These approaches include: Annotation of methods and variables and Type Inference [11]. Stephan L. in [4] and Ivan E. in [11] presented in their respective works the use of method and variable annotation as a means of indicating the types of variables used, in order to automate the test data generation process. With type inference, to deduce the variable type at compile time, the program must contain at least some static type annotations or a partial type. From this point onwards, for a program that contains no annotations, it will be very difficult to deduce variable types.

We introduce here some tools for automatically generating unit test data include. Pyngui [4] is an automatic unit test generation tool for the Python language. As limitations, it does not consider field declarations, assignments and collections (arrays, lists, dictionaries). RuTeG is a test generator for producing tests for the Ruby language [15]. However, this tool only generates simple test cases. It is up to the user to write the specific code to generate the data. On top of this, using a search-based approach is very resource-intensive. SymJS is a tool for automating unit tests for the JavaScript language [16]. However, this tool is not fully automatic, as the user must write code snippets in the form of annotations to provide information on the types of variables used in the target program, in order to help the tool generate the data that will satisfy the evaluation criteria. In

addition, this tool does not allow test data to be generated at API (Application Programming Interface) scale.

### C. Proposed Approach

Figure 1 shows the architecture of our approach. As input (1), we have a method under test. The method is then analyzed (2) by an analyzer that provides probable variables types. Then test values are randomly generated (3). In (4), we make proposals to the tester to validate the generated data. If he validates the data, we rebuild or recompose the method parameters with the new values (5). Finally, in (6) we generate the test method.

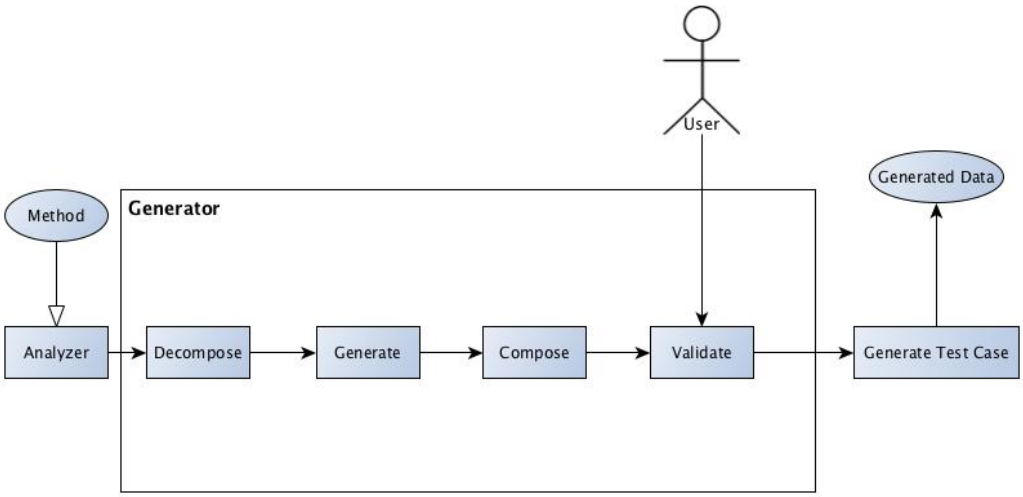


**Figure1 .** AuGenDa Architecture

As shown in Figure 1, the approach is divided into two main stages: the analyzer and the generator.

Analyzer: This component analyzes the program, extracting information that will be used to generate test data. Lexical analysis consists in breaking down the method into several lexical elements. Each element is called a token. All tokens are analyzed according to the following categories: punctuation, keywords, operators, literals and identifiers. Identifiers, for example, contain the names of variables and functions. Next, we perform a syntactic analysis to reorganize the tokens according to the above-mentioned categories. An abstract syntax tree is then created. The Abstract Syntax Tree (AST) is used to retrieve the program's constants and variables, and the relationships between these tokens. In effect, these are the leaves of the AST. For each identified variable, an oracle returns the type of the current variable. At the end of this step, we'll have information on the constructors, methods and arguments used in the program, as well as the various paths and conditions used.

The generator: The generator produces the data that will be transmitted as input arguments to the methods. The real challenge here is to find values that match the user's requirements. To do this, once we have information on the constructors and variables used in a method, we'll randomly generate the data types that will satisfy the conditions set as input and compare them with the output.

The approach is implemented by the algorithms below :

```
Input : M (a Method )
Ouput : Tm ( test Data)
Declaration : L
Begin
   switch kind (M)
      Getter  :TestGetter(M)
      Setter  : TestSetter (M)
      Other  : TestMethod(M)
END
```

**Algorithm AuGenDa**

This is the main algorithm. It takes a method as input and returns test data as output. We know that there are several types of method: getters, setters and normal methods. Once we've received the method to be tested as input, we start by checking the type of the method. Generation will be based on the method's type.

```
Input       : M (a Method)
Output      : Tm (test Data)
Declaration :        Param    (parameters list )
                     class (M class)
                     L (instance variable list)
                     LT (values list)
                     NP
                     NPARAM (new parameters generated)
Begin
   Param <- getParam (M)
   class <- getClass(M)
   For each p in Param do :
                L <-decompose ( p )
         LT <-mutate (L)
         NP <-compose ( class, LT)
         NPARAM <- add (NP)
  EndFor
  generateTestData (NPARAM, class
 End
```

**Algorithm TestMethod**

This algorithm takes a method as input. First, we retrieve the parameters of the method to be tested. For each parameter (input), we'll break it down into its basic type (instance variable). Once we've decomposed all the parameters, we'll mutate them. Mutation here consists in randomly generating data according to the base type obtained in the decomposition step. Then, using the compose method, we'll recompose the parameters received as input to the method, this time with the generated data to form a new object. Then, we'll call the generateTestData method, which will take the generated data as input and write the test method.

```
Input       : p (an object)
Ouput : L (object list)
Declaration          :NP
Begin
  p isLitteral ifTrue : NP add p
```

```
                ifFalse : NP adds [each e in p.instancevariables : decompose(e)]
return NP
End
```

**Algorithm Decompose**

The Decompose algorithm provides variables with a basic type. Let's suppose that in the main algorithm, we receive as input a method that will take as parameter a person (surname, first name and date of birth) to generate, for example, the price of a ticket for a trip to a travel agency. The aim here is to test this method. The method takes as input a person with a surname, first name and date of birth. In TestMethod, we'll retrieve the parameters (last name, first name and date of birth). For each element in the parameter list, we'll decompose it. As a result, we'll have a String for the name, another String for the first name and a Date for dateBirth. From there, it's easy for us to generate the random data for these parameters by calling the mutate method. Once that's done, we'll compose a new object with the newly generated data and, finally, generate a test method by calling generateTestData.

```
Input      : L ( a list of instance variables)
Ouput : listOfValue (          generated values)
Declaration : listOfValue
Begin
  For       i =1 to length(L) do
     listOfValue (i)<-L(i ) atRandom
  endFor
  return  listOfValue
 End
```

**Algorithm Mutate**

As mentioned above, the Mutate algorithm takes as input a list of atomic parameters (an integer, a string, a date, etc.). For each parameter, it generates a random value. Finally, it returns a list of the same length with newly generated values.

```
Input      : class (a class)
listOfValue (generated values)
Ouput :NP (new parameters)
Declaration :NP
Begin
    NP<-instantiate class with listOfValue.
    return NP
End
```

**Algorithm Compose**

The Compose algorithm will allow us to reconstruct test data from the generated values. We generate the test data with the generateTestData method.

```
Input : Param (parameters)
          class ( the class of the Method)
Ouput : Tm (data)
Declaration : newObject, (a class instance )
```

```
Begin
    newObject<-instantiate a class  object with Param
return newObject
End
```

**Algorithm generateTestData**

The generateTestCase algorithm is used to write the unit test with the currently generated values without constraints. This algorithm uses the values generated in the previous step.

```
Input      :newObject
Ouput : testCase
Begin
 System.AllMethods.Add<- TestClass.Method(
  self assert: Class.Method(Object) equals: newObject )
End
```

**Algorithm GenerateTestCase**

## D. Results and discussion

In this section, we empirically evaluate our data generation approach using a realistic case study. Our evaluation aims to answer the following questions: **RQ1**: Does our approach generate data? Question 1 asks whether our approach can generate data. As in the case of tests, there are three possibilities: the method produces nothing, the method produces erroneous data and the method produces an expected result. To do this, we're going to generate several different types of data and then analyze the results. **RQ2**: Do our approaches generate data in an acceptable time? Since the approach uses a random generator, is the data produced in an acceptable time? in other words, doesn't the developer waste a lot of time waiting? **RQ3**: Can our approach generate data that are both valid and representative? Question 3 aims to determine whether our approach produces suitable data. Given that the approach applies no validity constraints, does the data generated by the approach meet the validity requirement?

We have implemented our approach in Pharo, a dynamically typed object-oriented programming language licensed by MIT. Inspired by Smalltalk, Pharo offers a simple-to-use, stable, robust, reflexive and immersive environment. The project's source code is available at http://smalltalkhub.com/mc/FaouziElMansour/AuGenDa/main. We carried out our experiments with Pharo version 9.0.0 on a standard personal computer processor core i5-2540M with 2.60x4 GHz frequency, 06 GB RAM running on 64-bit Ubuntu 18.04.5 LTS.

For RQ1, we chose three Pharo packages. We generated test data for each method in the package

**Table 1.** Test data generation

| Package | #methods | #success |
|---|---|---|
| AST-Core | 1 422 | 1 413 |
| Kernel | 5 022 | 4 987 |
| Iceberg | 1 588 | 1 547 |

**RQ1** : Table 1 shows that the approach generates test data in 98,94% of cases. For the cases that failed, we note that these are methods specific to the Pharo language. These methods are traits and extension methods. These two types of methods manipulate objects that are not of the same class as the one in which they are implemented. This indicates that our methods work for general rules but not for specific cases, and therefore need to be adapted to each language.

The answer to RQ1 is that our approach generates test data. However, it needs to be specialized for language-specific techniques.

**Table 2.** Execution times

| Packages | #total methods | # total times | min | max | mean |
|---|---|---|---|---|---|
| AST-Core | 1 422 | 79 632 | 0 | 2 348 | 56 |
| Kernel | 5 022 | 431 892 | 0 | 4 233 | 86 |
| Iceberg | 1 588 | 68 284 | 0 | 3 277 | 43 |

**RQ2**: for RQ2 we estimated the time taken to generate each data item in ms. We used the utility provided by Pharo to monitor the time per method. In Pharo stops at primitive calls or calls that are too fast to be benchmarked (~ < 3ms). We have found that for many methods, the time required is below the 3ms threshold. To eliminate this difficulty, we launched profiling on all the methods in the package. Table 2 present the results. According to these results, the minimum time is under the threshold of 3ms; and the maximum time is 4233ms for the experiment. The average, which is a parameter that summarizes these data, is respectively 56 for AST-Core, 86 for Kernel and 43 for Iceberg. These average times is obtained by dividing the total time by the number of data items generated. We conclude that this time is reasonable for data that needs to be validated by a human being.

The answer to RQ2 is that the approach generates data within a reasonable time. For complex systems, parallelization can be envisaged. Similarly, we can generate data in advance as soon as the method is compiled.

**RQ3**: Can our approach generate data that are both valid and representative? Question 3 aims to determine whether our approach produces suitable data. Given the approach applies no validity constraints, does the data generated by the approach meet the validity requirement.

This research question concerns the representativeness and validity of the data generated. For this we required 3 experts. We randomly selected 300 methods from our experiments; for each package we had 100 generated data. Each expert was given these 300 generated data to examine. They were asked to comment on the validity and representativeness of the data. The results are shown in Table 3.

**Table 3.** Experts evaluations

| | Valid | | Representativeness | | | |
|---|---|---|---|---|---|---|
| | yes | No | None | Little | medium | Quite |
| Expert 1 | 236 | 64 | 64 | 27 | 186 | 23 |
| Expert 2 | 278 | 22 | 22 | 69 | 197 | 12 |
| Expert 3 | 227 | 73 | 73 | 31 | 169 | 27 |

According to the experts' judgement in Table 3, the data generated are 82,33% valid and, for the most part, fairly representative.

The answer to RQ3 is that our approach generates valid and representative data.

In short, we have demonstrated the feasibility of the proposed approach and its effectiveness. The next section presents the threats to validity.

## E. Threats to validity

The results of our empirical experiment are subject to threats of validity. The conclusions drawn and the validity of the data from the approach and implementation are the most relevant aspects of validity for our case study. The following notable threats are listed:

1. Validity of the approach. We carried out the analysis on a single programming language - this threatens the general validity of our findings. Generalization is always a concern in case studies, especially when results are drawn from a single case. The language studied may not be representative of dynamically typed languages. Further studies remain essential to determine how our approach will work on other languages. We are convinced that our approach is adaptable to any dynamically typed language.

2. Data validity. As indicated in the introduction, our approach allows test input data to be generated. The unavailability of test case output data prevented us from using the data generated by our approach for system-level testing. To mitigate this threat, we validated our generated data with domain experts. We therefore believe that the probability of major omissions in our data schema is low.

3. The experts' assessments of non-calculable measures in our experiment may be subjective. However, we also believe that the experts who made these assessments are experienced, which makes our study credible.

4. Internal threats to validity are related to the implementation of the approach. For example, we performed random generations of values. Consequently, it is always possible that our implementation of the approach contains errors that could affect the accuracy of our results. To counter this threat, we manually examined a subset of the results and found no apparent errors.

## F. Conclusion

Software testing consumes a lot of resources, but doesn't add any functionality to the product. We have proposed an approach to reduce this effort through Automatic generation of test data. In particular, it is a difficult task for dynamically typed languages, as there is no typing information. Various methods of automatically generating test data have been introduced. The main problem in the process of generating test data is to determine which data is valid. This work aims to improve the automatic generation of unit test data for dynamically typed languages. It is a relevant topic because its objectives are to facilitate the software testing phase. The proposed approach is simple and effective approach to generating test data. The approach is effective, functional and adaptable. It ultimately increases developer productivity by reducing errors during the testing phase and work time. It also improves software quality. Future works will extend this approach to other dynamically typed programming languages. Next, we plan to

define and use constraints to generate test cases or combine our approach with existing ones.

## G. References


[1] M. A. UMAR, "Comprehensive study of software testing: Categories, levels, techniques, and types," techRXiv.org, 2020.

[2] B. MEYER, "Seven principles of software testing," *Computer,* vol. 41, no. 8, pp. 99-101., 2008.

[3] Y. YANG, M. HE and M. LI, "Phase distribution of software development effort.," *ACM-IEEE Empirical software engineering and measurement,* vol. 2, pp. 61-69., 2008.

[4] S. LUKASCZYK, F. KROIß and G. FRASER, "Automated unit test generation for python," in *Proceedings 12. Springer International Search-Based Software Engineering: 12th International Symposium, SSBSE*, Bari, Italy,, 2020.

[5] V. GAROUSI, A. RAINER and P. LAUVÅS JR, "Software-testing education: A systematic literature mapping.," *Journal of Systems and Software,* vol. 165, p. 110570., 2020.

[6] A. DAMIA, M. ESNAASHARI and M. PARVIZIMOSAED, "Software Testing using an AdaptiveGenetic Algorithm.," *Journal of AI and Data Mining,* vol. 9, no. 4, pp. 465-474., 2021.

[7] M. ESNAASHARI and A. H. DAMIA, "Automation of software test data generation using genetic algorithm and reinforcement learning.," *Expert Systems with Applications,* vol. 183, p. 115446., 2021.

[8] E. NIKRAVAN and S. PARSA, "Hybrid adaptive random testing," *International Journal of Computing Science and Mathematics,* vol. 11, no. 3, pp. 209-221, 2020.

[9] R. HUANG, X. C. XIE and Y. Tsong, "Adaptive random test case generation for combinatorial testing.," *IEEE Annual Computer Software and Applications Conference.,* vol. 36, no. IEEE, pp. 52-61, 2012.

[10] D. S. RODRIGUES, M. E. DELAMARO and C. G. CORREA, "Using genetic algorithms in test data generation: A critical systematic mapping.," *ACM Computing Surveys (CSUR),* vol. 51, no. 2, pp. 1-23., 2018.

[11] I. ENDERLIN, " Génération automatique de tests unitaires avec Praspel, un langage de spécification pour PHP," Thèse de doctorat. Université de Franche-Comté., Franche-Comté, 2014.

[12] G. I. LATIU, O. A. CRET and L. VACARIU, "Automatic test data generation for software path testing using evolutionary algorithms," in *Third International Conference on Emerging Intelligent Data and Web Technologies*, 2012.

[13] D. HOLLANDER, "Automatic unit test generation," Université de technologie de Delft, Delft, 2010.

[14] Y. SUN, M. WU and W. RUAN, "Concolic testing for deep neural networks," *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering,* vol. 33, pp. 109-119., 2018.

[15] R. F. e. R. T. Stefan Mairhofer, "Search-based Software Testing and Test Data Generation for a Dynamic Programming Language," in *>. In : Conference : 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings*, Dublin, 2011.

[16] H. TANIDA, T. UEHARA and G. LI, "Automated unit testing of JavaScript code through symbolic executor SymJS.," *International Journal on Advances in Software,* vol. 8, no. 1, pp. 146-155., 2015.